John Catalana
Professor Kanemoto
CPSC-39
7/18/2025

Final Programming Report Project
Project Title: Terminal Dungeon Adventure Game

What the Game Does:

This is a text-based dungeon game that runs in the console. You play as a character, and in this case you're named the "Hero", and you explore different rooms connected like a maze. You can move around in the directions north, east, south, west as long as you type it in the console. You can pick up items, fight monsters, and try to find the exit. The game is all typed commands so there are no graphics or UI. It's fun because you have to think about which room to explore and remember which direction you're coming from. You also get to fight monsters with the help of the items you collect. It's a simple game but fun nonetheless.

Algorithms Used:

1. Monster Chain Attack (Recursion)
   This algorithm lets monsters attack the player after the player attacks the monster. It checks if there are any monsters left, makes each attack in order, and it'll either stop when the player dies or when all monsters are defeated.
   Steps:
   a. If the queue of monsters is empty, stop the function.
   b. Peek at the first monster.
   c. Print a message about the monster.
   d. Ask the player if they want to attack or run.
   e. If they attack
      - Damage the monster
      - If defeated, remove from queue
      - Call the function again (recursion)
   f. If run: exit the function

```
private void fightMonsters(Room room) {
    Queue<Monster> monsters = room.getMonsters();
    Scanner scanner = new Scanner(System.in);

    System.out.println("You've encountered enemies!");

    while(!monsters.isEmpty()) {
```

```
            Monster monster = monsters.peek();
            System.out.println("A " + monster.getName() + " appears with " +
monster.getHealth() + " HP. ");

            System.out.println("Commands: attack, run");
            String input = scanner.nextLine();

            if (input.equals("attack")) {
                monster.takeDamage(10); // Player does flat 10 damage
                System.out.println("You hit the " + monster.getName() + " for
10 damage");

                if (!monster.isAlive()) {
                    System.out.println("You defeated the " + monster.getName()
+ "!");
                    Item drop = monster.getDropItem();
                    monsters.poll(); // remove defeated monster
                    if (drop!= null) {
                        System.out.println("It dropped a " + drop.getName() +
"!");
                        Room currentRoom = dungeonMap.get(currentRoomId);
                        if (currentRoom.getItem() == null) {
                            currentRoom.setItem(drop);
                        } else {
                            System.out.println("But there's already an item
here, so the drop disappears");
                        }
                    }
                } else {
                    // Recursive monster retaliation
                    System.out.println("The monsters retaliate!");
                    Queue<Monster> copy = new LinkedList<>(monsters);
                    monsterChainAttack(copy,0);
                }
            } else if (input.equals("run")) {
                System.out.println("You fled the room!");
                return;
            } else {
                System.out.println("Unknown command.");
            }
        }
```

```
// Recursive function that lets all monsters attack one by one
    private void monsterChainAttack(Queue<Monster> monstersCopy, int
monsterIndex) {
        if (monstersCopy.isEmpty()) {
            return; // base case: no more monsters
        }
        Monster monster = monstersCopy.poll();
        if (monster.isAlive()) {
            int damage = monster.attack();
            player.takeDamage(damage);
            System.out.println(monster.getName() + " attacks you for " + damage
+ " damage!");
            System.out.println("Your HP: " + player.getHealth());
        }
        // Recurse into next monster
        monsterChainAttack(monstersCopy, monsterIndex + 1);
    }
```

2. Inventory Print (Recursion)
   This prints all the items the player has by checking each item one by one using a recursive function until the whole inventory is shown.
   Steps:
   a. Start with index 0
   b. If index equals inventory length then stop
   c. If an item exists at this index, print it.
   d. Call the function again with index + 1

```
    public void printInventoryRecursive() {
        System.out.println("Inventory:");
        printInventoryHelper(0); // Start recursion at index 0
    }
    // Helper method for resursive inventory printing
    private void printInventoryHelper(int index) {
        if (index >= inventorySize) {
            return; // Base case
        }
        System.out.println("- " + inventory[index].getName());
        printInventoryHelper(index + 1); // Resursive step
    }
```

3. Undo Movement (Stack)

   When the player moves to a new room, the previous room is saved on a stack. If the player types “undo,” the game pops the last room from the stack and moves the player back there.

   Steps:

   a. When moving, push the current room ID to the stack.
   b. When “undo” is entered, check if the stack isn’t empty.
   c. Pop the last room ID and set it as the current room.
   d. Notify the player they’ve returned to the previous room.

```
 switch (command) {
                case "move":
                    if (parts.length < 2) {
                        System.out.println("Please specifiy a direction (north,
south, east, west).");
                        break;
                    }

                    String direction = parts[1];
                    String nextRoomId = currentRoom.getExit(direction);

                    if (nextRoomId == null) {
                        System.out.println("You can't go that way.");
                        break;
                    }

                    roomHistory.push(currentRoomId); // Save where we were
                    currentRoomId = nextRoomId; // Then update where we're
going

                    Room nextRoom = dungeonMap.get(currentRoomId);
                    System.out.println("\n" + nextRoom.getDescription());
                    nextRoom.printExits();

case "undo":
                    if (roomHistory.isEmpty()) {
                        System.out.println("No previous room to return to.");
                    } else {
                        currentRoomId = roomHistory.pop(); // Go back one room
                        Room backRoom = dungeonMap.get(currentRoomId);
                        System.out.println("You backtracked to the previous
room");
                        System.out.println(backRoom.getDescription());
```

```
                        backRoom.printExits(); // shows the avaiable exits in
that room
```

How I Created the Algorithms and Used ChatGPT:

When building this game, I made sure to design the algorithms myself based on how I wanted the gameplay to work. I didn't just copy code but thought about the game logic first and how I was going to break it down into steps.

For example, in the Monster Chain Attack, I knew I wanted all the monsters in a room but they were going to attack the player one by one and not all at once. Instead of using a for loop, I wanted to challenge myself and use recursion but also to meet the requirements. I thought through how recursion works and decided that if there were no monsters left, the function should stop. Otherwise, the current monster should attack, and then the function should call itself again. I wrote it out in steps then turned that into code.

For the inventory print system, I wanted to do something more interesting than just a for-loop. A recursive version would be cleaner and show that I can use this method in different scenarios. I was printing them in a for-loop at first but then that's when I changed it. So I started by printing the first item, then called the same function with the next index.

For the Undo System, I figured that a stack would be perfect. It follows the idea of "last-in, first-out". So every time the player moves, I push the current room ID onto the stack. Then if a player were to type in "undo". I pop the last room from the stack and send the player back.

But while working on these, I did use ChatGPT as a helper. For example, I asked how Java handles stacks and queues, and chatGPT was able to explain the use of Stack and LinkedList. I also asked for advice on how to structure my recursive functions. The advice was super helpful because it assisted me into writing the code around my game's specific design. ChatGPT was a tutor to me and was pointing me in the right direction. It also helped me debug my code when I couldn't figure out why I was having errors or I was having a logic issue. Creating these algorithms was one of the most rewarding parts of this project because it was so satisfying to see everything run in its intended use and how everything connected.

Big-O Time Complexity:

Understanding Big-O time complexity helped me analyze how efficient my code is, especially for parts of the game that repeat often like the movement, combat, or inventory aspects.

1. Monster Chain Attack (Recursion) – O(n)
   This function goes through each monster in a queue and has them attack one by one using recursion. The function is called once per monster, so the total number of steps grows linearly with the number of monsters in the room. That's why the time complexity is O(n), where *n* is the number of monsters.

2. Inventory Print (Recursion) – O(n)
   This function prints the player's inventory items recursively. It checks and prints one item at a time, and then moves on to the next index. It repeats this until all items are printed, so it's also O(n), where n is the number of items in the inventory.

3. Undo Movement Stack – O(1)
   The undo feature uses a stack to keep track of visited rooms. When the player moves, the room ID is pushed onto the stack. When "undo" is typed, the last room is popped off the stack. Both push() and pop() are constant time operations, so this feature is O(1).

4. HashMap Room Lookup – O(1)
   I used a HashMap to store rooms with unique IDs. When the player moves or when the game needs to look up a room, it uses the ID as the key. Hash maps give average-case constant time access (O(1)), which keeps things fast even as the number of rooms grows. These time complexities show that even as the game grows with more monsters or rooms, key features like movement and combat stay efficient and responsive.

Data Structures Used and Explained

Choosing the right data structures was important to make the game run efficiently and to organize the code clearly.

1. Array – Player Inventory
   I used a simple array to store the player's inventory. Arrays are fixed in size, which made it easier to limit how many items the player could carry (like a backpack with a certain number of slots). It also allowed me to directly access items by index when using them or printing them.

2. LinkedList (Queue) – Monster Turn Order
   I used Java's LinkedList class as a queue to manage the monsters in each room. This lets monsters take turns attacking in the order they were added. I chose LinkedList because it allows efficient removal of the front element (like poll()) and it was easier to add new monsters.

3. Stack – Undo/Backtrack System
   When the player moves, I push their previous room ID onto a Stack. If they want to backtrack, the last room ID is popped and the player is sent back. This follows the Last-In, First-Out (LIFO) pattern, which is exactly how a stack works.

4. HashMap – Room Lookup by ID
   Each room in the game is stored in a HashMap<String, Room> using a unique string ID as the key. This allows the game to instantly find the right room when the player wants to move, without searching through a list. This was important for fast movement and keeping the game responsive.

5. Strings – Commands and Descriptions
   User input is read as strings (like "move north" or "use potion"), and the game parses these to decide what to do. I also used strings for item names, room descriptions, and other text displayed to the player.

6. Classes (Records / OOP Structure) – Game Entities
   I used custom classes (Player, Monster, Item, Room) to represent game elements. These acted like structured records to store multiple related values together, like a monster's name, health, and attack power. This made the code cleaner.

Challenges and How I Fixed Them:

At first, I got errors when moving between rooms because sometimes the next room aws just null. I fixed this problem by checking if the room exists even before moving. I also had some problems with variables that were named the same inside functions, so I renamed them to avoid confusion. Lastly, I made sure the inventory didn't overflow by limiting how many items the player can carry.

Also, when working on the recursive monster attack, at first, it was difficult. I didn't include a solid base case so as a result, it kept calling itself endlessly and the program crashed. I learned with recursion, it's important to check for a stopping condition. I fixed it by adding a check to see if the monster queue was empty before calling the function again.

There was a bug where items dropped by monsters would disappear or overwrite other items in the room. I fixed this by checking if a room already had an item before placing the dropped item there. If an item was already present, the new one would not be added and a message would be shown to the player. This made the item system feel more realistic and it prevented weird bugs from happening.

I also didn't handle mixed-case commands well. At first, a player had to type exactly how the command was like "move north" or the game wouldn't understand the command. I just fixed this by trimming the input and converting it to lowercase before parsing, and by adding an error message when there was a command that wasn't recognizable.

What I Would Add Next Time:

Although I'm really happy as to how the game turned out so far, there's still a lot I'd like to add if I had more time.

1. Locked Doors and Key-Based Puzzles
   One Idea I really wanted to add is having certain doors or paths locked unless the player was able to find a key or solve a puzzle. I feel like this would add more depth to the gameplay and make the game have more features. It also encourages players to explore the rooms more thoroughly. For example, a room might contain a treasure but it's blocked until a player finds a special item elsewhere in the dungeon

2. Shops and NPCS
   It would be so cool to introduce a merchant NPC where the player can trade items or buy items. This could add another use for collected items and even implementing a currency system. NPCs could also give side quests or just helpful tips on how to play the game.

3. Equipment and Stats System
   Right now, the player only has health but it would be fun to let players equip weapons and armor and they all give different bonuses or certain features. I could also add stats like strength, defense, or speed to make the combat system more complex but also make fights more strategic.

Conclusion:

Building this dungeon game was super fun but also a great learning experience. I was able to implement all the different things we've learned this summer and it helped me practice

working with different data structures, writing recursive functions, and organizing code using OOP. I faced quite a bit of challenges along the way because I haven't made something this big before but breaking down each part of the game into smaller sections made the process easier. I'm proud of how the game turned out. It's playable, fun, and showcased what I've learned in this class so far. There's still a lot I want to improve on in the future, but overall, this project gave me a better understanding of how to design and build something from scratch using real programming concepts.